

Amendments to the Specification:

Please replace paragraphs [0003], [0016], [0027], [0029], [0032], [0038], [0039], [0042], [0043], [0053], [0055] and [0056] with the following amended paragraphs respectively:

[0003] The amount of memory storage to store the symbol table information may be large. Some compilers may have a limit on the amount of valid symbol table information that can be used. Examples of such compilers include Java JAVA compilers. A static Java JAVA compiler converts Java JAVA source code into a verifiably secure and compact architecture-neutral intermediate format, call Java JAVA byte codes. A Java JAVA Virtual Machine interprets the byte code instructions at run time. To improve run-time performance, a Just-In-Time (JIT) Java JAVA compiler produces native code from Java JAVA byte code instructions during program execution. However, the symbol table information generated by the Java JAVA language is not sufficient to be used by the JIT compiler to provide debugging support. This is because the symbol table information in class files is generated for architecture independent byte codes while the JIT compiler generates machine specific instructions.

[0016] In the following description, for purposes of explanation, numerous details are set forth in order to provide a thorough understanding of the present invention. However, it will be apparent to one skilled in the art that these specific details are not required in order to practice the present invention. In other instances, well-known electrical structures and circuits are shown in block diagram form in order not to obscure the present invention. For examples, although the description of the invention is directed to the Java JAVA debugging architecture platform, the invention can be practiced for other debugging architecture platforms having similar characteristics.

[0027] The debugger 210 is a process being debugged. The debugger 210 includes an application 212, a virtual machine 214, and a back-end 216. The application 212 is the user's application or program being debugged. The application 212 includes the debugging mechanisms for data access support to be described later. The virtual machine 214 is an abstract model instantiated for the platform to run the application 212. In one embodiment, the virtual machine 214 is a Java JAVA virtual machine (JVM) implementing the Java JAVA Virtual

Machine Debug Interface (JVMDI). The back-end 216 communicates with the front-end 230 via the communication channel 220. The back-end 216 receives requests from the front-end and passes the requests to the virtual machine 214. The back-end 216 sends response to the requests from the virtual machine 214 to the front-end 230. The communication channel 220 is the link between the front end 230 and the back end 216. The transport mechanism for the communication channel 220 may be any one of sockets, serial lines, and shared memory. In one embodiment, the format and semantics of the serialized bit-stream flowing over the channel 220 is specified by the Java JAVA Debug Wire Protocol (JDWP). The front end 230 sends requests from user to, and receives response from, the back end 216. In one embodiment, the front end 230 implements the high-level Java JAVA Debug Interface (JDI) and uses the information from the low-level JDWP. The user interface 250 provides interface to user for interactions with the application 212. The user interface 250 may be implemented by any vendor specific tool. In one embodiment, the user interface 250 is client of the JDI.

[0029] The compilation infrastructure 310 compiles a source code into optimized native code to be run on the virtual machine 214 (Figure 2). The compilation infrastructure 310 includes a fast code generator 330, a profile representation 340, and an optimizing compiler 350. The fast code generator 330 compiles methods in a byte code sequence 320 using the debug support code from the debug support 370 into un-optimized native code 335. In one embodiment, the fast code generator 330 and the optimizing compiler 350 are part of a Just-In-Time (JIT) compiler such as the Intel JIT compiler. The fast code generator 330 does not generate any explicit intermediate representation to translate Java JAVA byte codes to native instructions of the underlying processor (e.g., Intel IA-32 and Intel IA-64). Rather, the fast code generator 330 uses the byte codes themselves to represent expressions and maintains auxiliary data structures that are managed on the fly during compilation. Since the fast code generation takes only two-linear time traversals over byte codes to produce the native code, compilation is simple and significantly fast. For this reason, dynamic re-compilation can be used efficiently to provide debugging information instead of computing and storing data structures for symbol table information. The profile data representation 340 includes statistics of information during execution compiled and collected by a counter 345. The optimizing compiler 350 re-compiles

methods that are identified as hot methods based on the collected profiling information and generates the optimized native code 360.

[0032] The stack frame access support 410 provides support for stack frame accesses. For stack frame accesses, the JIT compiler provides a mechanism to identify and access the caller's frame context, referred to as unwinding stack frame. The stack unwinding process starts with a frame context of the caller. If the active frame belongs to a Java JAVA method compiled by JIT compiler (referred to as jitted method), the VM makes calls to the compiler to perform the unwinding operation. This is due to the fact that the compiler has all information about the layout of the frames. The information about the return *ip*, location and size of spill area, location for saving callee-save registers and outgoing arguments can be obtained from JIT compiler for jitted methods. However, if the active frame does not belong to any jitted method, the VM uses its internal data structures to find the caller's context.

[0038] The data break-point 440 provides users with a mechanism to stop the program execution in terms of the program's memory states. Data break-points are also referred to as watch points. Java JAVA is a type safe language that does not allow arbitrary memory updates in the heap. Therefore, JVMDI does not have the interface that monitors any arbitrary memory location but have the support for field watches. Data breakpoint is one of the crucial features for tracking or isolating bugs in software that are otherwise hard or even impossible to find. The data break-point support 440 includes a field access and modifications watch 445 to provide support for field watches.

[0039] The JVMDI specification for field watches requires *an event hook function* to be called every time the *watched* field is accessed or modified. Field access events are generated when the filed specified is about to be accessed. Field accesses from Java JAVA Language code or from JNI are watched. Field accessed by other means is not watched. Similarly, filed modification events are generated when the specified filed is about to be modified from Java JAVA Language code or from JNI. JVMDI also supports cancellation of field watches during program execution. For each event generated, the hook function is called with an argument describing the event type and additional information specific to the event.

[0042] The basic support for field watch is to insert an instrumentation code to the method's code space. This instrumentation code passes necessary information to a run-time library function, which calls the event hook function. To prevent significant runtime overhead caused by un-activated filed watches, the execution of the instrumentation code is guarded by modifying the method's address space or using Boolean flag for each Java JAVA class field.

[0043] The run-time library functions are native functions that have different calling conventions than the Java JAVA class methods. They also require different handling to keep track of the live root set for garbage collection. Due to these differences and some other bookkeeping operations performed by the JVM, the native methods are not directly called from the method's code space. Instead, they are called through a stub that prepares the *epilog* and *prolog*, and passes the arguments to the native method. Thus, for each instrumentation code inserted for field access or modification, there is a call to stub that calls the run-time library function. The JIT compiler generates spills for the operands on the mimic stack that are live across the call sites prior to the calls to the stub. Only the operands, such as floating point operands and local variables, that may be killed by the call sites are saved.

[0046] The compilation 505 compiles a function that is in byte code format to produce native code. In one embodiment, the function is a Java JAVA method. The compiled function occupies a code space.

[0047] The generation 510 generates an instrumentation code corresponding to a field watch of a field. In one embodiment, the field is a Java JAVA field and the field watch is a Java JAVA field watch as specified in the JVMDI. The generation 510 executes a field watch sequence (Block 520). For the static model, the execution of the field watch sequence is performed whether or not the field watch is activated. This field watch sequence includes the instruction sequence to spill the mimic stack operands, which are live at the field access point, to their canonical spill locations. For the semi-static model, the execution of the field watch sequence is performed when the field watch is activated. The execution of the field watch sequence 520 includes saving live global state (Block 522), executing an event hook function for an event corresponding to the field watch (Block 524), and restoring the live global state (Block

526). The live global state corresponds to an active register such as scratch registers or floating-point registers.

[0053] The compilation 601 is the first or initial compilation. It includes a compilation of byte code 605 and a guard of execution 610. The compilation of byte code 605 compiles a byte code sequence having a field byte code that accesses or modifies a field. In one embodiment, the field byte code may be one of a getfield, getstatic, putfield, and putstatic byte codes as defined by the JVM specification (e.g., The Java **JAVA™** Virtual Machine Specification, Second edition, by Tim Lindholm and Frank Yellin, published by Addison-Wesley, 1999).

[0055] The recompilation 602 recompiles a function including the byte code sequence when the field watch is activated. Since it is performed by the fast code generator, the execution is fast and efficient. The recompilation 602 includes the compilation of byte code 605, the guard 610; a generation of instrumentation code 630, and an insertion instrumentation code 650. The generation 630 is essentially similar to the generation 510 (Figure 5). It generates an instrumentation code corresponding to a field watch of the field. The generation 630 includes an execution 640 of a field watch sequence if the field watch is activated. The execution 640 includes saving live global state (Block 642), executing an event hook function for an event corresponding to the field watch (Block 644), and restoring the live global state (Block 646). The live global state corresponds to an active register such as a live scratch register or a floating-point register. In one embodiment, saving the live global state includes pushing the live global state onto a stack, and restoring the saved live global state includes retrieving it from the stack. The execution of the event hook function includes passing an argument corresponding to the field and calling a run-time library function related to the event (Block 645).

[0056] The insertion 650 is similar to the insertion 550 for a semi-static model. It inserts the instrumentation code to the native code. In one embodiment, the insertion 650 inserts the instrumentation code in a stub at the end of the code space of the method (Block 655).